Appendix1

Application Scenario Language

Application Scenario Language (ASL) is an XML-based language that describes application business flow in small scenarios. The scenario language constructs can be changed, improved, and extended; they are here in this form to illustrate the invention. Scenarios consist of XML elements: scenario steps or acts. Every act of a scenario is a prompt, a condition, or an execution step.

ASL Elements

1. The "prompt" element

The prompt element prompts a user or a partner program for an answer

The prompt might have additional arguments, specific rules for input interpretation, and conditional actions.

Here is an extract from the addKnowledge.xml scenario, which allows someone to introduce a new object to the knowledgebase.

```
  <prompt variable="NEW-OBJECT"
     service="com.its.connector.ScenarioPlayer"
     action="prompt"
     noinput="reprompt(Your input is needed)"
     translate="concatenate"
     msg="Please provide a name for your new topic." />
<if condition="!exists" perform="doNextStep(acceptNewObject)" />
<act name="reject" action="query" constant="NEW-OBJECT"
 lastMsg="NEW-OBJECT is not new." />
<act name="acceptNewObject" service="com.its.connector.KnowledgeService"
  action="createNewPermanent" constant="NEW-OBJECT" />
```

The prompt element of the scenario will invoke the prompt mechanism (method) of the ScenarioPlayer 200. The prompt() method of the ScenarioPlayer class works with the Presenter 400 component to deliver a prompt message. The method shifts the ScenarioPlayer into the interpretation state. It will interpret the user's response and assign the variable provided with the prompt parameters to the value of the interpretation result.

The prompt element of the XML scenario specifies the service-class name (com.its.connector.ScenarioPlayer) and the action-method name (prompt), and sets the prompt variable (NEW-OBJECT) to store the user's input. One of the most important arguments of the prompt element is the prompt message delivered to the target audience.

The noinput and translate elements are optional interpretation parameters. The noinput element directs the program to re-prompt a user if the user just pressed the ENTER key.

The translate element instructs the program to concatenate multi-word input into a single word that can better serve as a unique reference.

For example, a user's input, such as "Volga river," will be translated to "VolgaRiver".

An important part of the prompt element is its internal variable element. In this example the variable has the name "NEW-OBJECT". The resulting interpretation of the response will be assigned to a variable created on the fly with the "NEW-OBJECT" name. The ScenarioPlayer will then parse the current scenario to replace all occurrences of "NEW-OBJECT with its value.

The second step of the scenario is a condition. If the object name does not exist (!exists) in knowledgebase, the system will perform the

doNextStep(acceptNewObject) operation. This means a jump to a specific act with the name "acceptNewObject," which is the last instruction in the example. The third step of the scenario is performed only if the name suggested by a user is known to the knowledgebase. The third step will query the knowledgebase on this known subject, and present the existing information with the last prompt (the end of the scenario), which lets the user know that this object name is not news to the knowledgebase.

2. The "doNextStep" element

The doNextStep element usually has an argument that points to an act of a current scenario to be performed next. The doNextStep element changes the order of acts of a scenario performed. The usual order is sequential, according to the original scenario source. The doNextStep element is often used in conditional actions, like in the example above.

3. The "playScenario" element

The playScenario element usually has one or more arguments that point to a new scenario to play. Optional arguments can include a particular step of this scenario to enter and additional arguments for this scenario. A proper scenario will be retrieved from the knowledgebase 100 and executed. Two examples of the usage of the playScenario element follow:

```
 playScenario(newUser)
playScenario(findMovie,knownRate,G-rated)
```

In the second example, the findMovie scenario will start with the knownRate act and will carry the additional argument "G-rated".

4. The "aliases" element

The aliases element can be a part of any prompt instruction; it lists alternatives for interpreting the response to the prompt. The aliases element can list possible answers that can come in different forms but still represent the same concept.

For example, the aliases element can instruct the program to interpret various user input, such as "y" or "sure", as "yes". In other words, if the user's input matches one of the aliases, the original input will be replaced with the related key value and all of the following conditions (if any) will apply to the resulting key.

The following example shows the aliases element in the user prompt.

```
<prompt variable="YES-OR-NO-ANSWER"
  action="prompt"
  service="com.its.connector.ScenarioPlayer"
  msg="Are you a new user? (y/n)"
  aliases="yes|y|new|sure^no|n|old"
    />
```

Note that there are two sets of aliases in the example above. These two sets are separated by the "^" character, while aliases within each set are separated with the "|" character. There could be multiple sets in the aliases element.

5. The "translate" element

The translate element instructs the system to perform string manipulations, or in other words, to translate the response. The translate element includes arguments and class and method names that provide specific translations, for example, replaceAll(oldString, newString).

If the class name is omitted, some default class will play the role of the translator.

Here is an example using the translate element:

```
<prompt variable="PERSON-NAME" action="prompt"
  service="com.its.connector.ScenarioPlayer"
  msg="Please provide your name (First Last)"
  translate=
"concatenate(REPLACE-WITH-INPUT)^startWithUpperCase(REPLACE-WITH-INPUT)"
     />
```

The system will use the default translator class with the concatenate and startWithUpperCase methods to concatenate a user's input and make sure each word begins with the upper case. For example, if the input was "jeff zhuk" the first instruction will produce "jeffZhuk" and the second instruction will make it "JeffZhuk".

The resulting string will be saved as a private data member of the ScenarioPlayer class.

6. The "condition" element

The "condition" element is a rich combination of conditional and execution expressions.

Here is an extract from the login scenario.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<scenario name="login">
   <prompt variable="USER-NAME"
    service="REPLACE-WITH-USER-SERVICE" action="prompt"
    msg="Your name?"
    noinput="playScenario(newUser)"  />
  <if condition="!exists" pattern="USER-NAME"
      perform="playScenario(newUser)"  />
  <!-- Retrieve password question and answer from the knowledgebase -->
  <act queryResult="PASSWORD-QUESTION=PASSWORD-ANSWER"
  query="passwordOf(USER-NAME)" />
  <prompt " variable="PASSWORD-INPUT"
      service="REPLACE-WITH-USER-SERVICE" action="prompt"
      msg="PASSWORD-QUESTION" echo="*" />
 <!-- Check password match  -->
  <if condition="!equals" source="PASSWORD-INPUT" pattern="PASSWORD-ANSWER"
       perform="doNextStep(wrongPassword)" />
<!-- successful login: more acts of the scenario -->
```

The very first instruction is a prompt asking for the user's name. The reply will be assigned to a variable, and the ScenarioPlayer will then parse the current scenario to replace all occurrences of the "USER-NAME" with its value.

A conditional statement in the next instruction checks for the user's existence. The condition in this statement is "!exists" and the pattern is the value of the USER-NAME variable.

The program will query the knowledgebase to verify the login name's existence. If this name is not known to the knowledgebase (!exists), the operation playScenario(newUser) will be performed.

The list of conditions in the above Application Scenario Language implementation example includes several keywords.

exists [pattern] - returns true if a pattern exists in the knowledgebase

!exists [pattern] - the opposite of exists, returns true if a pattern does not exist in the knowledgebase

equals [source] [pattern] - compares a source to a pattern and returns true if the source equals the pattern
!equals [source] [pattern] - compares a source to a pattern and returns true if the source is not the same as the pattern
includes [source] [pattern] - returns true if the source includes the pattern
!includes [source] [pattern] - returns true if the source does not include the pattern
inAliases [source] [pattern] - looks for a pattern in a table of aliases (located in configuration files or the knowledgebase) where the source defines the name of the table of aliases; returns true if the pattern is found as one of aliases
!inAliases [source] [pattern] - looks for a pattern in a table of aliases (located in configuration files or the knowledgebase) where the source defines the name of the table of aliases; returns true if the pattern is not found in the table
The noinput option, which can be used in the prompt statement, will perform an instruction in the case when the user does not answer the prompt but simply presses the Enter key.
A condition statement is followed by an action to perform, an executable instruction. If a condition is not met (returns false) the action will not be performed, and the next scenario step will be played instead.
The instruction includes the name of a service operation (method) to be performed and (optionally) the name of the service (class). If the noinput instruction does not include a service name, the ScenarioPlayer is used by default.
7. Executable instructions
Both the perform and action executable instructions use a service object and an action-method name for this service method invocation. The only difference between the two is the default service name in the case when a service name is omitted in the statement.
The perform instruction would use the ScenarioPlayer class in this case, while the action instruction would target the KnowledgeService instead.
Executable instructions accept parameters in the form of a method signature (the method name followed by its parameters) or in the form of key-value elements. Here is an example of a signature approach:
perform="playScenario(scenarioName)"
This statement will perform the playScenario() method of the ScenarioPlayer. This method must expect a single string argument.
A more generic way is to provide all the necessary arguments as key-value elements.
<act service="com.its.jmail.EmailClient" action="sendMail"
 to=jeff.zhuk@JavaSchool.com subject="training"
 msg="Please send me your list of courses" />
The sendMail method of the com.its.jmail.EmailClient service expects a set of arguments, like "to", "subject", and "msg", provided, for example, in a Hashtable of key-values.
8. The "query" element
The query element consists of a knowledgebase query and usually includes the queryResult option that assigns variables to store query results.
<act queryResult="PASSWORD-QUESTION=PASSWORD-ANSWER"
 query="passwordOf(USER-NAME)" />
The query instruction provided in the example above checks in the

knowledgebase for a user's record and delivers one of the user's password questions with its answer. In the current example, knowledgebase records include more than one way to check a user's identity. There may be different password questions as well as password answers, and the knowledgebase would select one or more of them based on some rules established by your requirements.

For example, the question can be as simple as "Password?", or more complicated, like "What is your mother's maiden name?", etc. The question is retrieved along with the answer, and both are assigned to proper variable names.